Exemple de modèle de conception de visiteur

Les modèles de conception sont utilisés pour résoudre les problèmes qui se produisent dans un modèle, nous le savons tous, non? Nous savons également que **les modèles de conception comportementale** sont des modèles de conception qui identifient les modèles de communication communs entre les objets. L'un de ces modèles de comportement est le modèle de visiteur, que nous allons découvrir dans cet article.

Si vous avez travaillé sur une application qui gère de nombreux produits, vous pouvez facilement comprendre ce problème:

> "Vous devez ajouter une nouvelle méthode à une hiérarchie de classes, mais le fait de l'ajouter peut être pénible ou endommager la conception."

Donc clairement, vous **voulez qu'une hiérarchie d'objets modifie leur comportement mais sans modifier leur code source** . Comment faire ça? Pour résoudre ce problème, le modèle de visiteur entre en scène.

Sections de cet article:

Présentation du modèle de visiteur

Selon Wikipédia, le **modèle de conception** du **visiteur** est un moyen de séparer un algorithme d'une structure d'objet sur laquelle il opère. Un résultat pratique de cette séparation est la possibilité d'ajouter de nouvelles opérations aux structures d'objets existantes sans modifier ces structures. C'est une façon

de suivre le **principe ouvert / fermé** (l'un des **principes de conception SOLID**).

La flexibilité de conception ci-dessus permet d'ajouter des méthodes à n'importe quelle hiérarchie d'objets sans modifier le code écrit pour la hiérarchie. Un mécanisme de **répartition** plutôt **double** est utilisé pour mettre en œuvre cette fonction. La double distribution est un mécanisme spécial qui distribue un appel de fonction à différentes fonctions concrètes en fonction des types d'exécution de deux objets impliqués dans l'appel.

Concevoir les participants / composants

Les classes des participants dans ce modèle sont:

Visiteur - Il s'agit d'une interface ou d'une classe abstraite utilisée pour déclarer les opérations de visite pour tous les types de classes visitables.

ConcreteVisitor - Pour chaque type de visiteur, toutes les méthodes de visite, déclarées en visiteur abstrait, doivent être implémentées. Chaque visiteur sera responsable de différentes opérations.

Visitable - est une interface qui déclare l'opération d'acceptation. C'est le point d'entrée qui permet à un objet d'être «visité» par l'objet visiteur.

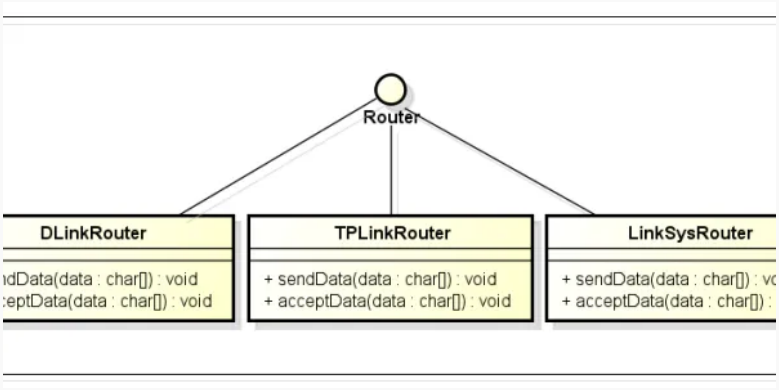
ConcreteVisitable - Ces classes implémentent l'interface ou la classe Visitable et définissent l'opération d'acceptation. L'objet visiteur est transmis à cet objet à l'aide de l'opération d'acceptation.

Exemple de problème à résoudre

Un exemple vaut toujours mieux qu'une longue théorie. Alors ayons-en un ici aussi pour le modèle de conception des visiteurs.

Supposons que nous ayons une application qui gère les routeurs dans différents environnements. Les routeurs doivent être capables d'envoyer et de recevoir des données char à partir d'autres nœuds et l'application doit être capable de configurer des routeurs dans différents environnements.

Essentiellement, la conception doit être suffisamment flexible pour prendre en charge les changements dans la manière dont les **routeurs peuvent être configurés pour des environnements supplémentaires à l'avenir, sans trop de modifications dans le code source** .



Routeurs existants dans l'application

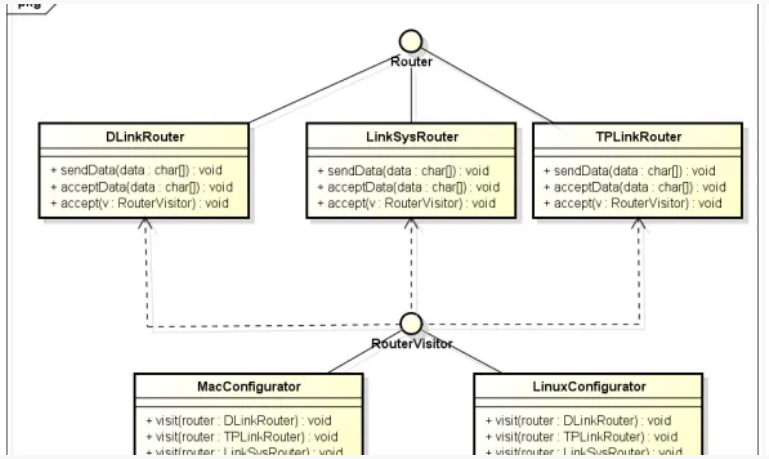
Nous avons plus de 3 types de routeurs et nous devons écrire du code pour eux. **Une façon de résoudre ce problème** consiste à définir des méthodes telles que configureForWinodws () et configureForLinux () dans l'interface Router.java et à les implémenter dans différents produits, car chacun aura ses propres paramètres et procédures de configuration.

Mais le problème avec l'approche ci-dessus est que chaque fois qu'un nouvel environnement est introduit, toute la hiérarchie du routeur devra être compilée à nouveau. Solution non acceptable. Alors, qu'est-ce qui peut empêcher cette situation?

Solution proposée utilisant le modèle de visiteur

Le modèle de visiteur convient parfaitement à ces types de problèmes dans lesquels vous souhaitez introduire une nouvelle opération dans la hiérarchie des objets, sans changer sa structure ni les modifier. Dans cette solution, nous allons implémenter **la technique de double envoi en introduisant deux méthodes, à savoir les méthodes accept () et visit ()** . La méthode accept () sera définie dans la hiérarchie du routeur et les méthodes de visite seront au niveau des visiteurs.

Chaque fois qu'un nouvel environnement doit être ajouté, un nouveau visiteur sera ajouté dans la hiérarchie des visiteurs et devra implémenter la méthode visit () pour tous les routeurs disponibles et c'est tout.



Solution utilisant le modèle de visiteur

Dans le diagramme de classes ci-dessus, nous avons des routeurs configurés pour les systèmes d'exploitation Mac et Linux. Si nous avons besoin d'ajouter la capacité pour Windows également, alors je n'ai pas besoin de changer de classe, il suffit de définir un nouveau visiteur WindowsConfigurator et d'implémenter les méthodes visit () définies dans l'interface RouterVisitor. Il fournira la fonctionnalité souhaitée sans aucune autre modification.

Code d'implémentation

Examinons le code source des différents fichiers impliqués dans le problème et la solution discutés ci-dessus.

Router.java

```
public interface Router
{
    public void sendData(char[] data);
    public void acceptData(char[] data);

    public void accept(RouterVisitor v);
}
```

DLinkRouter.java

```
public class DLinkRouter implements Router{

    @Override
    public void sendData(char[] data) {
    }

    @Override
    public void acceptData(char[] data) {
    }

    @Override
    public void accept(RouterVisitor v) {
```

```
        v.visit(this);
    }
}
```

LinkSysRouter.java

```
public class LinkSysRouter implements Router{

    @Override
    public void sendData(char[] data) {
    }

    @Override
    public void acceptData(char[] data) {
    }

    @Override
    public void accept(RouterVisitor v) {
        v.visit(this);
    }
}
```

TPLinkRouter.java

```
public class TPLinkRouter implements Router{

    @Override
    public void sendData(char[] data) {
    }

    @Override
    public void acceptData(char[] data) {
    }

    @Override
    public void accept(RouterVisitor v) {
        v.visit(this);
    }
}
```

RouterVisitor.java

```
public interface RouterVisitor {
    public void visit(DLinkRouter router);
    public void visit(TPLinkRouter router);
    public void visit(LinkSysRouter router);
}
```

LinuxConfigurator.java

```
public class LinuxConfigurator implements RouterVisitor{

    @Override
    public void visit(DLinkRouter router) {
        System.out.println("DLinkRouter Configuration for Linux
```

```
    }

    @Override
    public void visit(TPLinkRouter router) {
        System.out.println("TPLinkRouter Configuration for Linux
    }

    @Override
    public void visit(LinkSysRouter router) {
        System.out.println("LinkSysRouter Configuration for Linu
    }
}
```

MacConfigurator.java

```
public class MacConfigurator implements RouterVisitor{

    @Override
    public void visit(DLinkRouter router) {
        System.out.println("DLinkRouter Configuration for Mac co
    }

    @Override
    public void visit(TPLinkRouter router) {
        System.out.println("TPLinkRouter Configuration for Mac c
    }

    @Override
    public void visit(LinkSysRouter router) {
        System.out.println("LinkSysRouter Configuration for Mac
    }
}
```

Comment utiliser les visiteurs dans le code d'application

Pour utiliser la conception ci-dessus, utilisez les visiteurs de la manière ci-dessous. J'ai utilisé le code sous forme de cas de test JUNIT, vous pouvez changer le code comme vous vous sentez correct dans votre cas.

TestVisitorPattern.java

```
public class TestVisitorPattern extends TestCase
{
    private MacConfigurator macConfigurator;
    private LinuxConfigurator linuxConfigurator;
    private DLinkRouter dlink;
    private TPLinkRouter tplink;
    private LinkSysRouter linksys;

    public void setUp()
    {
        macConfigurator = new MacConfigurator();
```

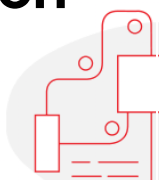
```
        linuxConfigurator = new LinuxConfigurator();

        dlink = new DLinkRouter();
        tplink = new TPLinkRouter();
        linksys = new LinkSysRouter();
    }

    public void testDlink()
    {
        dlink.accept(macConfigurator);
        dlink.accept(linuxConfigurator);
    }

    public void testTPLink()
    {
        tplink.accept(macConfigurator);
        tplink.accept(linuxConfigurator);
    }

    public void testLinkSys()
    {
        linksys.accept(macConfigurator);
        linksys.accept(linuxConfigurator);
    }
}

Output:

DLinkRouter Configuration for Mac complete !!
DLinkRouter Configuration for Linux complete !!
LinkSysRouter Configuration for Mac complete !!
LinkSysRouter Configuration for Linux complete !!
TPLinkRouter Configuration for Mac complete !!
TPLinkRouter Configuration for Linux complete !!
```

Téléchargement du code source

Pour télécharger le code source de l'application ci-dessus, suivez le lien donné.

 Téléchargement du Souececode

Bon apprentissage !!



A propos de Lokesh Gupta

Un gars de la famille avec une nature aimante et amusante. Aime les ordinateurs, la programmation et la résolution de problèmes quotidiens. Retrouvez-moi sur Facebook et Twitter .

Rétroaction, discussion et commentaires

Rishabh

6 avril 2018

qu'est-ce que la classe TestCase ici et où elle est créée et appartient à quel package ..

Muhammad

16 janvier 2016

Bonne conversation à tous! s'il vous plaît, je ne sais pas si quelqu'un peut m'aider à expliquer le modèle de conception des visiteurs.

Mladen

17 juillet 2015

Juste une pensée. Dans Java 8, vous pouvez utiliser la méthode par défaut dans l'interface du routeur, afin de ne pas avoir à répéter l'implémentation sur toutes les implémentations de routeur.

public interface Router

{

public void sendData(char[] data);

public void acceptData(char[] data);

default public void accept(RouterVisitor v) {

v.visit(this);

}

}

> Lokesh Gupta
>
> July 17, 2015
>
> Awesome thought !! A really useful update. Thanks for suggesting. But at this point, I would not like to update the existing source because Java 8 still is not adopted in most of the organizations for production (in my knowledge).

Marving

July 12, 2017

Nice post. Many Thanks.

One question. About comment Mladen

public interface Router

. . . . . .

default public void accept(RouterVisitor v)

{

v.visit(this); // at this moment this is type of is Router

}

And RouterVisitor hast NOT method visit(Router router)

public interface RouterVisitor {

public void visit(DLinkRouter router);

public void visit(TPLinkRouter router);

public void visit(LinkSysRouter router);

}

DLinkRouter is a Router

but

Router NOT is a DlinkRouter

NOT is a TPLinkRouter

NOT is a LinkSysRouter

What do you think about?

Rajendra J

May 14, 2015

Hi Lokesh,

I like all your topic discussions,

I want to know template pattern in detail (am writing here as it is also comes under behavioural pattern).

Please discuss about 'Template pattern' if you have some time.

Thanks in advance.

Lokesh Gupta

May 14, 2015

Right now, I am very busy in my other projects. I will soon update on this.

java luver

February 27, 2015

looked around for good explanation/usecase of visitor design pattern. I must admit this is far the best example/explanation. Kudos to the author!

Vishi

February 24, 2014

RouterVisitor should take Router interface as dependency. You are using the concrete Router classes which violates the SOLID DI principle..

so RouterVisitor should have only 1 method visit(Router router)

In Each Concrete Configurator class, we can implement visit method with Reflection (for all RouterVisitor types)

viru

November 3, 2013

There is no main method and TestCase class..

> Lokesh Gupta
>
> November 4, 2013
>
> public class TestVisitorPattern extends TestCase

Loganathan

October 24, 2013

Si nous avons introduit un nouveau routeur, nous devons à nouveau modifier tous les droits des visiteurs car la méthode visit () obtient la classe Visitable et non l'interface. Encore une fois, cela fera un gros problème dans le principe OUVERT / FERMÉ. pourquoi ne pas utiliser l'interface et imprimer le message en fonction du type d'objet. Corrigez-moi si j'ai tort, s'il-vous plait.

> Lokesh Gupta
>
> 25 octobre 2013

Oui, tu as raison. Un nouveau routeur devra être configuré pour chaque environnement et cela changera beaucoup de classes, mais n'est-ce pas ainsi que cela devrait être fait? Cela oblige le développeur à gérer un nouveau routeur dans chaque environnement pris en charge.

Amit Das

31 juillet 2019

Ne pensez-vous pas que si nous n'utilisons pas de visiteur ici, nous forçons également le développeur à gérer un nouveau routeur dans chaque environnement pris en charge.

Suresh

14 octobre 2013

Selon la définition, si nous modifions la structure d'un objet qui ne devrait pas affecter les autres... dans votre exemple, je dois ajouter une nouvelle méthode dans la classe DLinkRouter, comment allons-nous gérer cela...?

Lokesh Gupta

14 octobre 2013

La définition est «ajouter une nouvelle méthode à une hiérarchie de classes». Pas pour un seul cours.

sasidhar

9 septembre 2013

votre explication est simple et tout le monde comprendra. Merci!

Vitaly

9 septembre 2013

Dans l'image, vous avez 2 routeurs LinkSys et pas de DLink

Lokesh Gupta

9 septembre 2013

Corrigé.

Les commentaires sont désactivés sur cet article!